

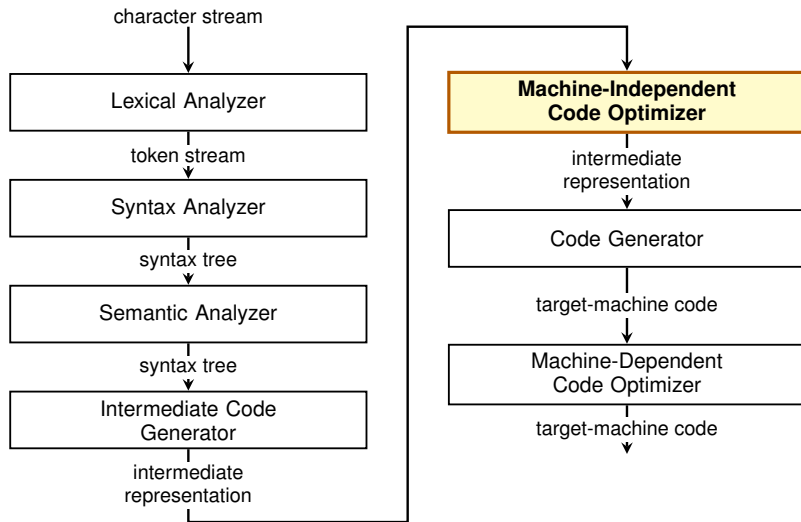
Basic Block Optimizations

KC Sivaramakrishnan

IIT Madras

Monsoon 2026

The Compiler



Overview of Optimizations

- Optimizations are program transformations that seek to improve a program's resource utilization
 - Execution time (most often)
 - Space
 - Code size
 - Network messages sent, etc.
- Optimizations should not alter what the program computes.
 - The observable behaviour of the program must stay the same.

Classification of Optimizations

For imperative languages like C, C++, Java, etc. there are three granularities of optimizations

- ① Local optimizations
 - Apply to a basic block in isolation
- ② Global optimizations
 - Apply to a control-flow graph (of a method) in isolation
- ③ Inter-procedural optimizations
 - Apply across method boundaries.

Most compilers do ①, many do ②, few do ③.

Cost of Optimizations

- In practice, a conscious decision is made not to implement the fanciest optimization known.
- Why?
 - Some optimizations are hard to implement.
 - Some optimizations are costly in compilation time.
 - Some optimizations have low benefit.
 - Many fancy optimizations are all three!
- **Goal:** Maximum benefit for minimum cost
- The term 'program optimization' is a slight misnomer: we don't necessarily get the 'optimal' code.
 - Program improvement is a more appropriate term.

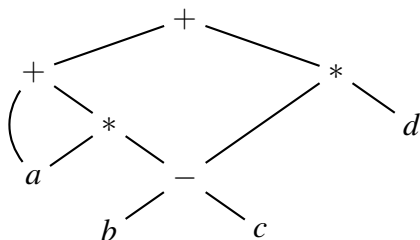
Local Optimizations

- The simplest form of optimizations.
- No need to analyze the entire procedure code, just look at a basic block.
 - It is a linear piece of code.
 - Analyzing and optimizing is easier.
 - Has local scope - and hence effect is limited.
- In spite of being simple, it can often provide substantial benefits.

DAG representation of basic blocks

Recall: DAG representation of expressions

- leaves corresponding to atomic operands, and interior nodes corresponding to operators.
- A node N has multiple parents - N is a common subexpression.
- Example: $(a + a * (b - c)) + ((b - c) * d)$



DAG construction: follow the current values

Process statements in order:

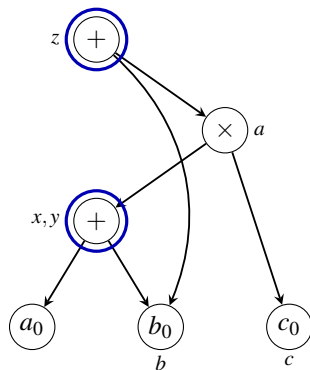
1. $x = a + b$
2. $y = a + b$
3. $a = x * c$
4. $z = a + b$

Leaves: values on entry, such as a_0, b_0, c_0 .

A variable label names its current value.

Assume only y and z are live on exit.

Blue rings mark outputs.



Outputs: retain the values of y and z and their dependencies.
The $\times$ node is still needed for z , even though a is not live on exit.

DAG construction: the rule behind each step

For an assignment $x = y \text{ op } z$:

- 1 **Read the operands first.** Find the nodes currently labeled y and z .
- 2 **Reuse or create.** Reuse a node with the same operator and the same ordered children; otherwise create one.
- 3 **Update the destination.** Remove x from its old node and attach it to the result node.
- 4 **Mark outputs at the end.** Nodes carrying live-out variables are outputs; their dependencies must also be retained.

Check: in $x = x + 1$, which x supplies the operand?

The old value: read operands before moving the destination label.

What can we do with the DAG?

Transformation

What to look for

Common subexpression elimination

Same operator and operand values: compute once, reuse the result.

Dead code elimination

A value with no remaining user and no live-out label: remove it.

Copy propagation

Two names for one value: use an existing name directly.

Algebraic simplification

A cheaper expression with the same permitted behaviour.

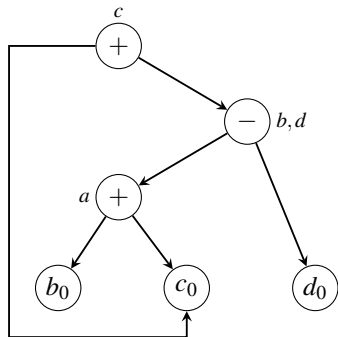
Our next question: how do we turn shared value nodes back into assignments?

CSE: two statements, one value

1. $a = b + c$
2. $b = a - d$
3. $c = b + c$
4. $d = a - d$

Question: do lines 2 and 4 use the same values of a and d ?

Yes. The RHS of line 4 reads the old d , before its destination is updated.



Result: four assignments, but only three arithmetic nodes.

From a shared node back to code

Original

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Compute each value once

```
a = b + c
d = a - d
c = d + c
b = d   if b is live out
```

The shared subtraction is stored in *d*. Both later uses can read it.

Why the copy? Sharing a computation does not remove the obligation to supply every live-out variable.

Children must be available before their parents are computed; preserve input values until their last use.

Same value does not always mean same DAG node

$$a = b + c$$

$$b = b - d$$

$$c = c + d$$

$$e = b + c$$

By value tracking:

$$a = b_0 + c_0$$

$$e = (b_0 - d_0) + (c_0 + d_0)$$

Can the last line reuse a ?

The two $+$ nodes have different children.
Structural CSE will not merge them.

By algebra: $e = b_0 + c_0 = a$, for exact integer arithmetic. An additional algebraic rewrite can expose the reuse.

This reassociation is not generally valid for floating point or arithmetic that traps on overflow.

Dead code: prune values with no remaining use

$a = b + c$

$b = b - d$

$c = c + d$

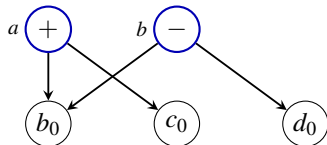
$e = b + c$

Assume only a, b are live out.

Remove c next. Keep a, b and the input values they depend on.

Repeat: delete a node with no users and no live-out label.

Do not delete c_0 : the live result a still depends on it.



Algebraic simplification: state the conditions

For already-computed scalar operands:

Candidate rewrite

$$x + 0 \rightarrow x, x \times 1 \rightarrow x$$

$$x \times 0 \rightarrow 0$$

$$0/x \rightarrow 0$$

$$a \&\& \text{true} \rightarrow a$$

When is it valid?

Ordinary integer arithmetic. Floating point needs its own rules.

Integer value computation; do not erase an effectful evaluation of x .

Integer division with $x \neq 0$; preserve required exceptions.

Boolean a ; preserve evaluation and short-circuit behaviour.

Check: can we replace `read() * 0` by `0`?

The value is zero, but the input operation may still be required.

Strength reduction for unsigned integers

Assume unsigned integers and no overflow.

Multiply by a power of two

$$x \times 8 \longrightarrow x \ll 3$$

Shift left by three bits.

Divide by a power of two

$$x/8 \longrightarrow x \gg 3$$

Use a **logical** right shift: fill the leftmost positions with zeros.

Example: unsigned $40/8 = 5$.

$$00101000 \longrightarrow 00000101$$

Use the shift when it is cheaper on the target machine.

Signed division: arithmetic shift is not enough

Question: can we replace signed division by two with an arithmetic right shift?

Try $x = -3$, with division rounding toward zero.

Signed division

$$-3/2 = -1$$

Rounds toward zero.

Arithmetic right shift

$$\text{ashr}(-3, 1) = -2$$

Copies the sign bit; rounds toward $-\infty$.

The results differ. Do not replace signed division by a plain arithmetic shift in general.

The rewrite needs an additional justification, such as a proof that x is nonnegative.

Reassociation can expose a common expression

Assume exact integer arithmetic and unchanged b, c, d .

Before

```
a = b + c
t = c + d
e = t + b
```

Regroup, then reuse

```
e = (c + d) + b = (b + c) + d
a = b + c
e = a + d
```

Remove the assignment to t only if t has no other use and is not live out.

Why the assumption matters: in finite-precision floating point, $(10^{20} + (-10^{20})) + 3$ can differ from $10^{20} + ((-10^{20}) + 3)$.

Copy propagation: how long does a copy stay valid?

Safe use

```
a = b  
x = 2 * a
```

⇓

```
a = b  
x = 2 * b
```

A source redefinition

```
a = b  
b = 7  
x = 2 * a
```

If old $b = 3$, then $a = 3$.

Replacing a by b changes x from 6 to 14.

Rule: after $a = b$, substitute b for a only while neither a nor b has been redefined.

Propagation can expose folding and CSE; liveness determines whether the copy itself can be removed.

Constant propagation makes folding possible

Input

$a = 5$

$x = 2 * a$

$y = x + 6$

$t = x * y$

Substitute values, then evaluate

$a = 5$

$x = 2 * 5 \rightarrow x = 10$

$y = 10 + 6 \rightarrow y = 16$

$t = 10 * 16 \rightarrow t = 160$

Propagation replaces an operand by a known constant.

Folding evaluates an operation on constants at compile time.

If only t is live out, dead code elimination leaves $t = 160$.

A sequence of passes: optimization opportunities

`a = square (x)`

`b = 3`

`c = x`

`d = c * c`

`e = b * 2`

`f = a + d`

`g = e * f`

Assume pure integer operations, no overflow, and only *g* live out.

`square (x)` denotes $x \times x$.

Find:

- a known constant;
- two computations that will become identical;
- names that may disappear.

One transformation enables the next

$a = x * x$

$b = 3$

$c = x$

$d = a$

$e = 6$

$f = a + a$

$g = 6 * f$

1. Expand the square

Expose the multiplication $x \times x$.

2. Propagate and fold

$c = x$ exposes a second $x \times x$; $b = 3$ makes $b \times 2$ constant.

3. Eliminate the common expression

Reuse a instead of multiplying again.

4. Propagate again

The new copy $d = a$ changes f ; $e = 6$ changes g .

Next: which assignments are still needed to produce g ?

Dead code elimination: trace back from the output

`a = x * x`

`b = 3`

`c = x`

`d = a`

`e = 6`

`f = a + a`

`g = 6 * f`

Only g is live out.

- 1 g needs f .
- 2 f needs a .
- 3 a needs the input x .
- 4 Nothing needs b, c, d, e .

Remove the four gray assignments.

Check: if d were also live out, would `d = a` survive? Yes.

The final block and the reason to iterate

Final code

`a = x * x`

`f = a + a`

`g = 6 * f`

Seven assignments became three.

Propagation exposed CSE.

CSE created a copy.

Propagation made assignments dead.

Repeat useful passes until they make no further change, or until a compilation budget is reached.

This is a local fixed point for the chosen rules, not a proof that the block is optimal.

Check with $x = 2$: both versions produce $g = 48$.

Array loads depend on memory as well as the index

```
x = a[i]
```

```
a[j] = y
```

```
z = a[i]
```

Can we replace the last line
with $z = x$?

Try $i=j$:

Initially $a[i] = 4$, and $y = 9$.

The first load gives $x = 4$.

The store changes $a[i]$ to 9.

The second load gives $z = 9$.

Same text, different value: the store may change the memory read by the second load.

If the compiler proves the locations do not overlap, reuse may be safe. Otherwise, keep the load.

A store invalidates a load's reuse

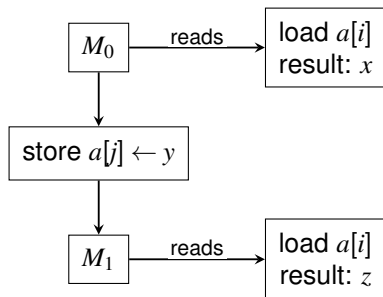
$x = a[i]$

$a[j] = y$

$z = a[i]$

M_0 : memory before the store.

M_1 : memory after it.



Conservative DAG rule: the store kills earlier load nodes that may depend on the modified location for future CSE.

Killed does not mean erased: x still holds the value loaded before the store.

Different base names can refer to the same array

Assume byte-address arithmetic, 4-byte elements, and initially $a[5] = 4$.

$b = a + 12$

b points to element 3 of a .

$x = a[5]$

$\text{addr}(b[2]) = a + 12 + 4 \times 2$

$b[2] = 9$

$\text{addr}(a[5]) = a + 4 \times 5$

$z = a[5]$

Both addresses are $a + 20$.

The store through b changes $a[5]$.

Question: can the last line become $z = x$?

No: $x = 4$, but $z = 9$. Replacing the second load by $z = x$ would give the wrong value.

Consequence: a store through b invalidates reuse of the earlier load through a . Tracking only the base name is insufficient.

Peephole optimization: a small instruction window

Generated code

MOV R1, R1

ADD R2, 0

MUL R3, 8

Candidate replacement

remove self-copy

remove add-zero

SHL R3, 3

Check the machine semantics: register widths, condition flags, overflow, and the cost of the replacement.

Example hazard: `ADD R2, 0` may set flags used by the next conditional branch.

A peephole is a small window over instructions; a DAG optimization works with the block's value dependencies.

A local rewrite can reveal another one

Before

```
MOV R1, R1
```

```
JMP L1
```

```
L1:  ADD R2, 0
```

```
NEXT
```

- 1 Remove the self-copy.
- 2 Remove the jump to the immediately following instruction.
- 3 Remove add-zero if its flag effects are dead.

Assume these rewrites preserve all machine effects. Recheck the changed neighbourhood after each replacement.

Keep `L1` if another branch still targets it. Nonadjacent matches also require checking the intervening instructions.

Redundant store: what can actually be deleted?

```
LOAD  a, R0
```

```
STORE R0, a
```

The store writes back the value just loaded.

Remove the store if both execute together, with no intervening change, and the memory access has no observable store effect.

Keep the load if a later instruction needs the value in R0.

Two checks:

- Can a branch enter directly at the store?
- Is the memory volatile, atomic, device memory, or otherwise observable?

Deleting the pair additionally requires a dead register result and a safely removable load.

Unreachable instructions: follow the incoming edges

No entry to the increment

```
JMP L2
```

```
INCR R0
```

```
L2:  NEXT
```

A possible other entry

```
JMP L2
```

```
L1:  INCR R0
```

```
L2:  NEXT
```

The left increment can be removed. On the right, another branch may target L1.

CFG connection: code is unreachable when no path from the entry reaches it. Being after a jump is only a local clue.

Jump threading: bypass a jump to a jump

Before

```
if cond goto L1
```

```
...
```

```
L1:  goto L2
```

After

```
if cond goto L2
```

```
...
```

```
L1:  goto L2
```

Why safe? The block at `L1` performs no work before jumping to `L2`. The same idea applies to an unconditional jump.

Can we now delete `L1`? Only after checking all its incoming edges, including fall-through.

Machine rewrites need a cost model

Pattern	Candidate	Check
Multiply by 8	Shift left by 3	Width, overflow and flags
Unsigned divide by 8	Logical shift right by 3	Unsigned interpretation
Signed divide by 8	Shift plus rounding correction, if needed	Rounding toward zero
Add zero	Remove instruction	Are condition flags dead?

Legal does not imply profitable. Compare latency, code size, register pressure, and the target's available instructions.

A practical rewrite procedure

- 1 **Match** a candidate instruction pattern.
- 2 **Check legality:** values, memory effects, flags, and control-flow entries.
- 3 **Check profitability** for the target machine.
- 4 **Replace and revisit** the affected neighbourhood.

Use a decreasing cost or a rewrite budget to avoid cycles. Stop when no selected rule applies; optimality is not guaranteed.

Exit question: why can an instruction still be needed when its destination is not live out?

Its value may feed a live result, or its execution may have observable effects.